4 Gewinnt KI

Hochschule Koblenz | KI | Louis Watermeyer (552364)

Inhalt

Das Spiel 4 Gewinnt

Das Spiel wird auf einem senkrecht stehenden hohlen Spielbrett gespielt, in dem die Spieler abwechselnd ihre Spielsteine fallen lassen. Das Spielbrett besteht aus sieben Spalten (senkrecht) und sechs Reihen (waagerecht). Wenn ein Spieler einen Spielstein in eine Spalte fallen lässt, besetzt dieser den untersten freien Platz der Spalte.

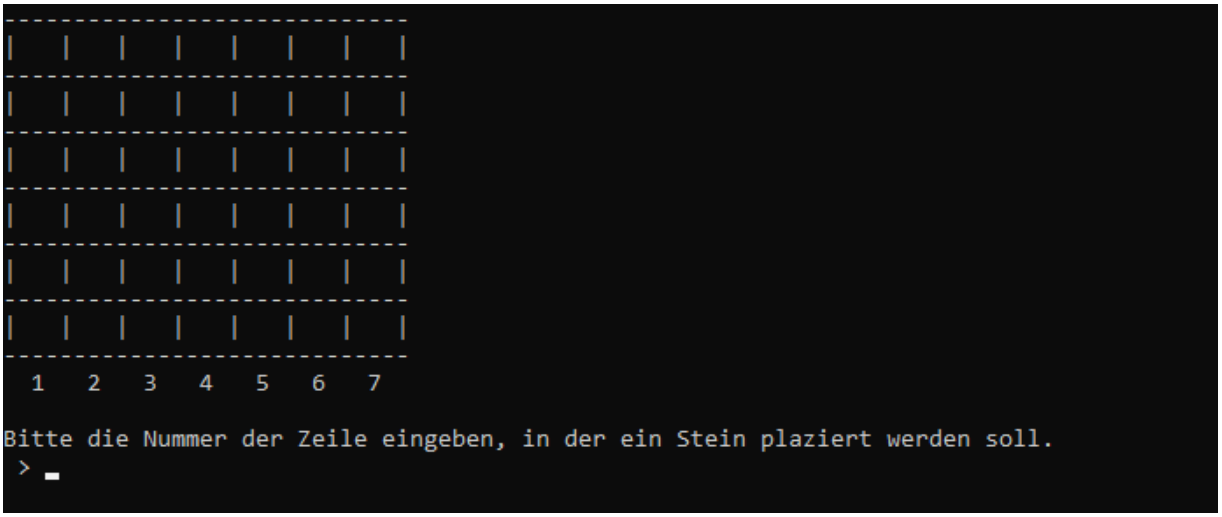
Siegbedingung

Gewinner ist der Spieler, der es als erster schafft, vier seiner Spielsteine waagerecht, senkrecht oder diagonal in eine Linie zu bringen. Das Spiel endet unentschieden, wenn das Spielbrett komplett gefüllt ist, ohne dass ein Spieler diese Siegbedingung erfüllt.

Spielumgebung

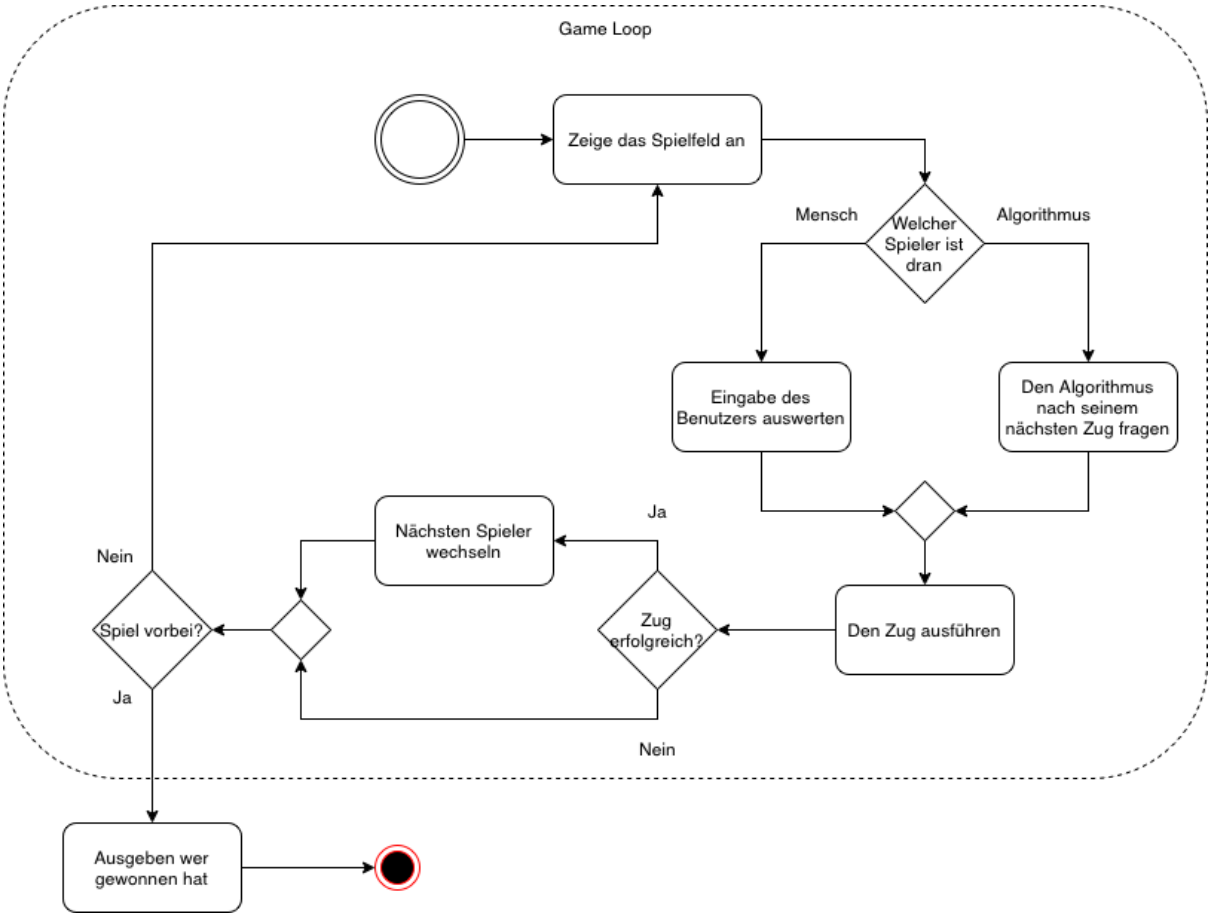
Der Spieler gibt in eine Konsole ein, wo er seinen Stein platzieren möchte. Die Steine der Spieler werden durch ein ‘X’ oder ein ‘O’ dargestellt. Sobald der Spieler einen Zug macht, soll der Algorithmus so schnell wie möglich mit einem sinnvollen Zug antworten. Das Spielfeld soll danach komplett neu aufgebaut werden, es soll also kein neues darunter ausgegeben werden, wie das bei Konsolenspielen oft der Fall ist, sondern der Output der Konsole soll gelöscht werden, um das neue Spielbrett anzuzeigen. Wenn ein Zug nicht möglich ist, sollte die Eingabe einfach ignoriert werden. Der entsprechende Spieler ist dann nochmal an der Reihe. Genau so soll das Spiel sich ebenfalls verhalten, wenn der Benutzer eine nicht zulässige Eingabe tätigt, wie z.B. einen Buchstaben oder eine Zahl, die höher ist als die Anzahl der Spalten.

Beispiel:



Programmumgebung

Das Programm wurde in C++ geschrieben , da ich bereits Erfahrung mit dieser Programmiersprache habe.
Benutzt wurden nur die C++ Standardbibliotheken mit Ausnahme der OpenMP Bibliothek, die es mir erlaubt, sehr leicht Aufgaben auf mehrere Threads zu verteilen.
Das Spiel selbst läuft in einer einzelnen Konsolenanwendung, die den menschlichen Spieler nach seinem Zug fragt, diesen ausführt und dann den von dem Algorithmus berechneten Zug spielt. Dieser Ablauf wiederholt sich so lange, bis entweder der Spieler oder der Algorithmus gewonnen hat. Das grobe Ablaufschema sieht wie folgt aus:



Wichtige Klassen

GameMaster

Eine Instanz dieser Klasse regelt das Spiel. Sie verwaltet das Spielfeld, das auf der Konsole angezeigt wird und entscheidet, wie die Schleife, in der das Spiel stattfindet, sich verhält. Dieser Klasse werden nacheinander die Züge der beiden Spieler gegeben.

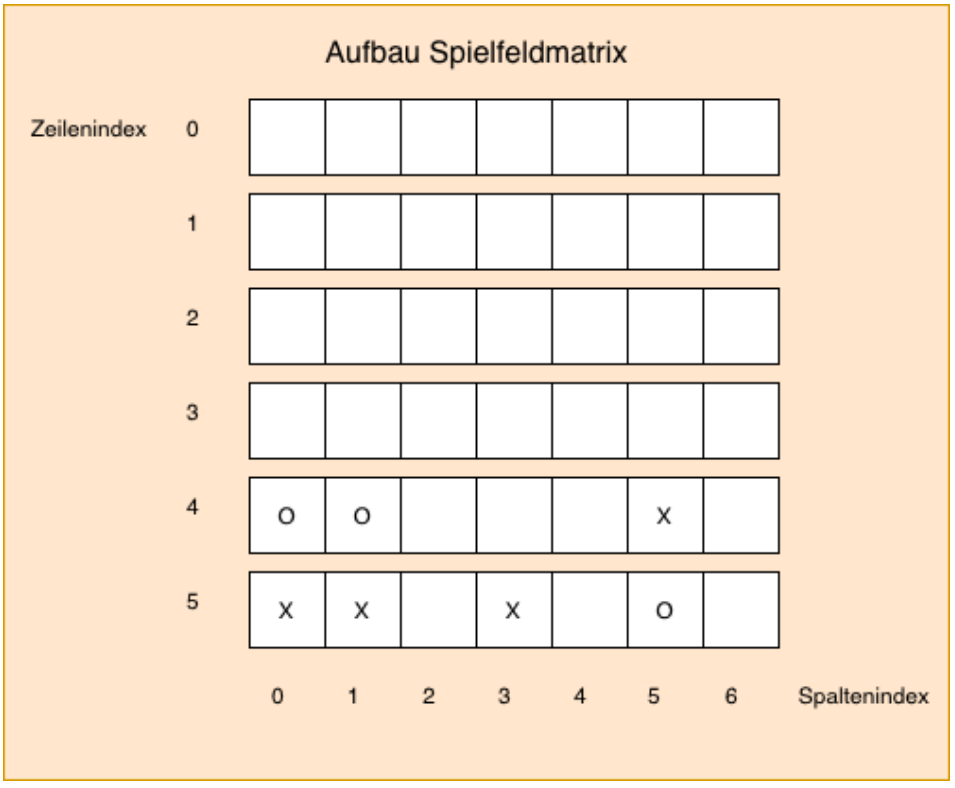
Field

Ein Field repräsentiert ein Spielfeld. Die Hauptfunktion dieser Klasse ist es, den Zustand eines Spielfeldes zu speichern und ein paar Hilfsfunktionen anzubieten, um es zu analysieren.

Ein Spielfeld wird intern als eine zweidimensionale Matrix aus "std::vector<char>" gespeichert. Die inneren Vektoren sind dabei die Reihen des Spielfeldes.

Die Klasse bietet Hilfsfunktionen wie „Field::getRow(int)", mit denen es erleichtert wird, das Spielfeld auszuwerten. Die Zeilen und Spalten werden wie im Bild indexiert.

Diese Klasse enthält auch die Logik, um zu prüfen, ob das Feld einen Sieg beinhaltet. Dieser Vorgang wird mehrere tausend Male durchgeführt, sollte also sehr Performance sein. Mehr zu diesem Ablauf im Kapitel “Optimierung der Siegbedingung”.



Node

Eine Node ist ein einzelnes Element im Suchbaum. Es besitzt ein Spielfeld, das ihren Zustand beschreibt und kann dieses evaluieren. Nach außen bietet die Node einen Wert an. Dieser wird ihr entweder zugeordnet, oder sie berechnet ihn selbst, indem das Spielfeld bewertet wird. Innerhalb des Minimax-Algorithmus berechnen die Endpunkte des Suchbaumes ihren Wert selbst und speichern diesen. Alle Nodes, die keine Endpunkte sind, bekommen den Wert des besten oder schlechtesten Nachfolgers zugeordnet, je nachdem welcher Spieler am Zug ist. So wird der Suchbaum von unten nach oben bewertet. Mehr dazu im Kapitel “Bewerten des Suchbaumes”.

Damit nachvollzogen werden kann, welche Züge der Algorithmus ausführen muss, um zu einem gewünschten Endzustand zu kommen, müssen die Nodes auch speichern, was der Unterschied zu ihrem Vorgänger ist, bzw. welcher Zug gemacht wurde, als sie erstellt wurden.

Das Problem mit den Nodes ist, dass diese zu tausenden erstellt werden. Somit werden auch sehr viele Matrizen erzeugt, die genutzt werden, um das Spielfeld zu speichern. Es dauert lange und verbraucht Speicher. Genauere Zahlen zur Geschwindigkeit und dem Speicherbedarf gibt es im Kapitel “Performance”.

Algorithm

Eine Klasse nach dem Singleton Pattern, die im Spiel als “zweiter Spieler” fungiert. Sie bietet Funktionen, die das Verhalten eines Spielers imitieren. Das Spiel wird innerhalb einer Schleife ausgeführt, in der abwechselnd der Spieler und die Instanz dieser Klasse nach Ihren Zügen gefragt werden. Innerhalb dieser Klasse gibt es einen Member „topNode“ der die Wurzel des Suchbaums darstellt. Dieser Suchbaum wird dann nach dem besten Zug durchsucht. Die Klasse bietet auch die Minimax-Funktion, die eine Node entgegennimmt und dort rekursiv den Minimax-Algorithmus anwendet. Nach dem Bewerten des Suchbaumes wird der nächste Zug gefunden, indem in den direkten Kindern des Wurzelknotens nach dem höchsten Wert gesucht wird. Aus dieser Node wird dann gelesen, welcher Zug nötig ist, um auf diesem Weg weiterzugehen.

Suchbaum aufstellen

Es wird ein Suchbaum aufgestellt, der ein paar Züge in die Zukunft sehen kann. Dazu wurde eine Klasse „Node“ erstellt, die einen solchen Zustand repräsentiert. Jede Node hat eine Liste von Kindern, die die nächsten Züge darstellen. Diese Liste wird von der Node rekursiv erstellt, bis entweder ein Endpunkt des Baumes oder die maximale Suchtiefe erreicht ist. Es werden nur Kinder erstellt, wenn der angeforderte Zug möglich ist. Das bedeutet, dass die meisten Nodes sieben Kinder haben, da meistens sieben Züge möglich sind. Es können aber auch weniger sein, wenn z.B. eine Spalte des Spielfeldes bereits voll ist.

Beschreibung der Funktion zum erstellen des Suchbaums

Der Suchbaum wird erstellt, indem auf den momentanen Wurzelknoten die Funktion Node::createNextMoves(int depth) aufgerufen wird.

```
/**
 * Creates the next possible moves recursively according to its field. These Moves will be represented by new nodes
 * which will be the children of this node.
 *
 * \param depth Is the maximum depth of the recursion.
 */
void Node::createNextMoves(int depth)
{
    // Don't create a deeper level if we bottom out here.
    if (depth == 0 || m_field.isGameOver())
        return;

    // If this node already has children, just pass the instruction along.
    // Otherwise create children.
    if (m_children.empty())
    {
        for (int nextMoveColumn = 0; nextMoveColumn <= m_field.width(); nextMoveColumn++)
        {
            Field::Player nextPlayer = m_turn == Field::Player::Human ? Field::Player::Algorithm
                : Field::Player::Human;
            if (m_field.isMovePossible(nextMoveColumn))
            {
                std::shared_ptr<Node> newChild = std::make_shared<Node>();
                newChild->init(m_field, nextPlayer, nextMoveColumn);
                m_children.push_back(newChild);
            }
        }
    }

    for (std::shared_ptr<Node> children : m_children)
    {
        children->createNextMoves(depth - 1);
    }
}
```

Darin wird als erstes die Abbruchbedingung der Rekursion geprüft. Es sollen keine weiteren Zustände erstellt werden, wenn die maximale Tiefe erreicht ist, oder das Spiel bereits beendet ist.
Wenn nichts davon zutrifft, muss dieselbe Funktion bei allen Kindern der Node aufgerufen werden. Sollte die Node noch keine Kinder haben, z.B. weil es der erste Durchlauf des Programms ist, oder sie vorher ein Endpunkt des Suchbaumes war, werden alle möglichen weiteren Züge erstellt und als Kinder dieser Node gespeichert. Das Erstellen dieser Kinder erweitert dann den Suchbaum.

Beim Erstellen der Kinder muss darauf geachtet werden, dass die Spieler sich abwechseln. Wenn die originale Node einen Zug des Menschen darstellt, müssen ihre Kinder einen Zug des Algorithmus simulieren.

Schlussendlich wird die Funktion rekursiv auf alle existierenden Kinder angewandt. Dabei wird der Parameter “depth” verringert, um die Rekursion am gewünschten Punkt zu stoppen.

Suchbaum bewerten (Minimax)

Nachdem der Suchbaum komplett aufgestellt wurde, wird er bewertet. Dazu wird die Minimax-Funktion der Algorithmus-Klasse genutzt. Diese nimmt eine Node entgegen und bewertet diese rekursiv. Dazu wird zuerst geprüft, ob das Spielfeld, das die Node beinhaltet, zu einem Sieg, einer Niederlage oder einem Unentschieden führt. Wenn das der Fall ist, müssen die weiteren Züge nicht bewertet werden und es kann verhindert werden, dass Spielfelder bewertet werden, die niemals passieren würden, da das Spiel vorher endet.

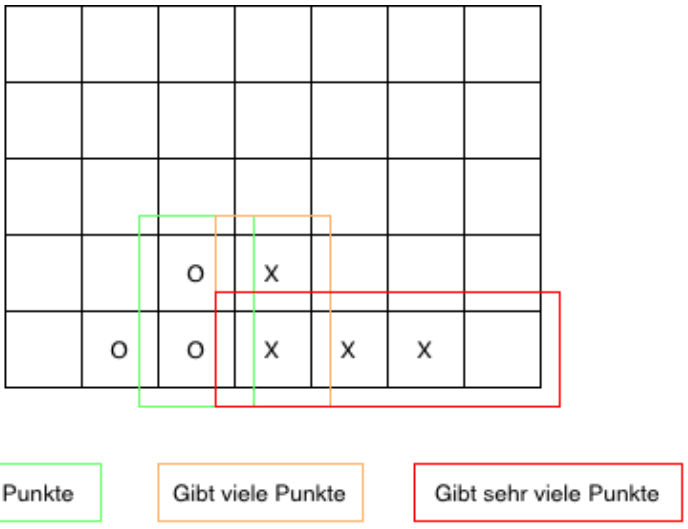
Wenn das Spielfeld kein Endpunkt ist, also Kinder hat, wird die Minimax-Funktion rekursiv auf diese angewandt. Wenn alle Kinder einen Wert zurückgegeben haben, wird der höchste oder der niedrigste Wert genommen (je nachdem welcher Spieler am Zug ist) und der momentanen Node zugeordnet. So werden die Werte nacheinander bis zum Wurzelknoten bestimmt.

Wenn die Node ein Endpunkt ist, wird das Spielfeld ausgewertet und dieser Wert wird zurückgegeben. So wird die Rekursion aufgelöst.

Das Spielfeld bekommt einen Wert zwischen -2147483648 (Verlieren) und 2147483647 (Sieg). Das sind die Grenzen eines int32. Ein Spielfeld, das Unentschieden ausgeht, hat den Wert 0.

Die Methode mit der das Spielfeld bewertet wird, achtet auf verschiedene Situationen:

- Spielsteine in der Mitte des Feldes sind mehr wert als Spielsteine an den Rändern, da von dort aus mehr Möglichkeiten für sinnvolle zukünftige Züge offen sind.
- Gruppen von gleichen Spielsteinen geben viele Punkte, oder ziehen viele Punkte ab. Je nachdem, welcher Spieler die Gruppe gebildet hat.
- Gruppen von gleichen Spielsteinen, die noch freie Nachbarfelder haben sind ebenfalls besonders wertvoll
- Enthält das Feld einen Sieg, eine Niederlage oder ein Unentschieden, wird ihm sofort der entsprechende Wert zugeordnet. Alle anderen Gruppen sind egal.



Nicht alle Gruppen sind aufgrund der Lesbarkeit markiert! Die Punkte, die für Spielsteine des Gegners vergeben werden, wirken sich natürlich negativ auf die Gesamtpunktzahl aus.

Diese Konstellationen werden in der Horizontalen, der Vertikalen und in beiden diagonalen Richtungen gesucht. Die Bewertung des Spielfeldes ist leider relativ aufwändig. Darum hatte das Alpha-Beta-Pruning wahrscheinlich einen sehr gut messbaren Effekt.Mehr dazu im Kapitel “Alpha-Beta-Pruning”.

Beschreibung der Minimax-Funktion

Die Minimax-Funktion der Klasse “Algorithm” wird wie folgt in Algorithm::getNextMove() aufgerufen:

```
// Evaluate tree
minimax(m_topLevelNode, TREE_DEPTH, INT_MIN, INT_MAX, Field::Player::Algorithm);
```

Die “topLevelNode” ist der Wurzelknoten des bereits erstellten Suchbaums. Dieser hat die Tiefe “TREE_DEPTH”. Die nächsten beiden Werte sind der Alpha- und Beta-Wert für das Alpha-Beta-Pruning. Der letzte Parameter gibt an, welcher Spieler als nächstes am Zug ist.

Innerhalb der Minimax-Funktion wird als erstes geprüft, ob die Node einen Endpunkt darstellt. Dies kann passieren, wenn die maximale Suchtiefe erreicht ist, oder wenn das Spiel, das die Node beschreibt, abgeschlossen ist. Wenn einer dieser Fälle eintritt, wird die Node bewertet und der errechnete Wert wird zurückgegeben.

Ist die Node kein Endpunkt, muss sie Kinder haben. Um der Node also einen Wert zuweisen zu können, muss für alle Kinder der Node ebenfalls die Minimax-Funktion aufgerufen werden. Diese Rekursion dauert so lange, bis ein oder die maximale Suchtiefe Endknoten erreicht wird.

Wenn alle Kinder einer Node einen Wert haben, muss entschieden werden, welchen Wert die originale Node bekommt. Dies hängt davon ab, welcher Spieler momentan am Zug ist. Ist der Algorithmus am Zug, muss die vielversprechendste Node gewählt werden. In unserem Fall bedeutet das, dass die originale Node den höchsten Wert annimmt, den eines ihrer Kinder hat. Ist der Mensch am Zug, nimmt sie den niedrigsten Wert.

Die Werte des Alpha-Beta-Prunings stellen den höchsten bzw. niedrigsten Wert dar, der bisher im Zweig zur Verfügung steht. Damit können wir ausschließen, ob wir einen Zweig zu Ende berechnen müssen. So können sehr viele Berechnungen durch logisches Ausschließen gespart werden. Die Grundidee ist, dass wir einen Zweig nicht zu Ende berechnen müssen, wenn wir bereits wissen, dass es einen besseren Weg gibt. Mehr dazu im Kapitel “Alpha-Beta-Pruning”.

So werden alle Nodes im Suchbaum von unten nach oben bewertet, bis wir wieder bei unserem Wurzelknoten ankommen. Sobald das passiert, ist der Minimax-Algorithmus abgeschlossen und es wird der nächste, vielversprechendste Zug getätigt.

Die “#pragma” Anweisungen sind für die genutzte Multithreading Bibliothek OpenMp. Hier werden die einzelnen Rekursionsinstanzen der Minimax-Funktion auf verschiedene Threads

aufgeteilt.So kann der Baum schneller bewertet werden. Das bedeutet jedoch nicht, dass jede Rekursion ihren eigenen Thread bekommt.

```cpp
/**
 * Minimax function that works recursively.
 *
 * \param node The node that gets evaluated.
 * \param depth The maximum search depth.
 * \param alpha Alpha value for Alpha-Beta pruning.
 * \param beta Beta value for Alpha-Beta pruning.
 * \param nextPlayer The player that makes the next move in reference to the given node.
 * \return
 */
int Algorythm::minimax(std::shared_ptr<Node> node, int depth, int alpha, int beta, Field::Player nextPlayer)
{
    // return the evaluation of a node if we have reached the maximum search depth.
    if (depth <= 0 || node->isGameOver())
    {
        node->evaluateState();
        return node->getNodeValue();
    }

    if (nextPlayer == Field::Player::Algorithm)
    {
        // Pick the best outcome
        int max = INT_MIN;

        #pragma omp parallel for
        for (int index = 0; index < node->getChildren().size(); index++)
        {
            max = std::max(max, minimax(node->getChildren()[index], depth - 1, alpha, beta, Field::Player::Human));
            alpha = std::max(alpha, max);

            // We don't need to check the rest of the children, if the human already has a better choice by taking
            // another branch.
            if (beta <= alpha)
                break;
        }
        node->setNodeValue(max);
        return max;
    }
    else
    {
        // Pick the worst outcome
        int min = INT_MAX;

        #pragma omp parallel for
        for (int index = 0; index < node->getChildren().size(); index++)
        {
            min = std::min(min, minimax(node->getChildren()[index], depth - 1, alpha, beta, Field::Player::Algorithm));
            beta = std::min(beta, min);

            // We don't need to check the rest of the children, if the algorithm already has a better choice by taking
            // another branch.
            if (beta <= alpha)
                break;
        }
        node->setNodeValue(min);
        return min;
    }
}
```

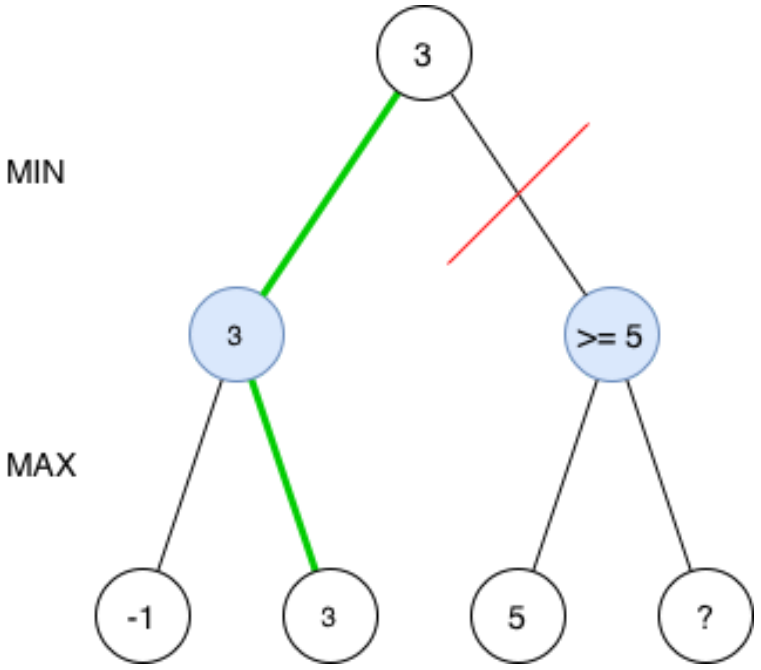
Multithreading

Um die Performance zu verbessern, wird die Minimax-Funktion auf mehrere Threads aufgeteilt. Dazu verwende ich eine Bibliothek namens OpenMP, die dynamisch abschätzt, was die Performanteste Lösung ist, um einen definierten Codebereich auf Threads aufzuteilen. So muss ich mich nicht darum kümmern, den verschiedenen Threads Aufgaben zuzuweisen. Die Bibliothek erkennt, dass die Minimax-Funktion rekursiv ist. Wenn ein Thread mit seiner Aufgabe fertig ist, bekommt er also eine neue Stelle im Baum zugeordnet. Des Weiteren ist diese Funktion bereits als Feature in der IDE, die ich benutze, eingebaut und kann einfach mit einem Parameter aktiviert werden.

Alpha-Beta-Pruning

Da die Bewertungsfunktion relativ aufwändig ist, macht es Sinn, Alpha-Beta-Pruning zu implementieren. Dazu wird in der rekursiven Minimax-Funktion geprüft, ob der momentan analysierte Zweig überhaupt evaluiert werden muss. So kann die Anzahl der Endknoten des Suchbaumes drastisch reduziert werden. Diese Nodes, die wir überspringen, müssen dann nicht die aufwendige Bewertungsfunktion ausführen.

Beispiel: Der weiße Spieler muss minimieren, der blaue maximieren.



Wir fangen links unten an, die Nodes zu bewerten und berechnen die Werte für -1 und 3. Der blaue Spieler übernimmt daraus die 3. Als Nächstes berechnen wir die Node, die den Wert 5 aufweist. Wir wissen nun, dass die blaue Node darüber mindestens den Wert 5 haben wird. Der weiße Spieler wird diesen Weg also am Anfang nie einschlagen, da er ja bereits mit dem linken Zweig eine für ihn bessere Alternative hat. Wir müssen die letzte Node also gar nicht bewerten. So können sehr viele Aufrufe der Bewertungsfunktion gespart werden. Das ist natürlich sehr gut, da diese Bewertungsfunktion relativ aufwändig ist.

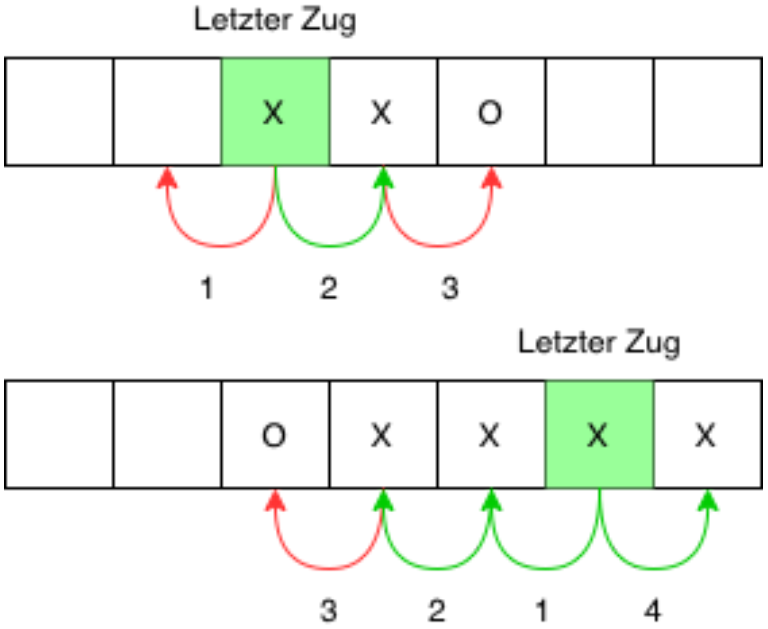
Das Pruning könnte noch performanter gemacht werden, wenn die Spielzüge so geordnet wären, dass ein wertvoller Spielzug zuerst ausgewertet wird. So würde sehr schnell ein hoher Wert gefunden und es werden viele Zweige übersprungen. In meiner Implementierung werden die nächsten Züge jedoch immer von links nach rechts erstellt. Das bedeutet, der erste Zug ist ein Stein in Zeile 1, dann 2, usw. Somit reize ich das Pruning nicht einhundertprozentig aus, aber es reicht für diesen Anwendungsfall.

Optimierung der Siegbedingung

Das Prüfen der Siegbedingung ist eine der am häufigsten durchgeführten Operationen. Darum sollte sie so performant wie möglich sein.

Mein Ansatz zu diesem Problem setzt voraus, dass der letzte Zug bekannt ist, und die Siegbedingung nach jedem Zug überprüft wird. So muss nur das Spielfeld in der direkten Umgebung des letzten Zuges geprüft werden. Dieser letzte Zug hat dann entweder zu einem Sieg geführt, oder nicht. Wenn nicht, können wir sicher sein, dass es auch im restlichen Feld keinen Sieg gibt, vorausgesetzt die oben genannten Bedingungen wurden eingehalten.

Beispiel:



Wenn der letzte Zug bekannt ist, kann das Feld links daneben geprüft werden. Wenn dieses Feld denselben Buchstaben enthält, wird in diese Richtung weiter geprüft, bis entweder kein Feld mehr übrig ist, oder ein nicht passender Buchstabe gefunden wird. Wenn das passiert, wird auf der rechten Seite weitergemacht. Auch dort wird abgebrochen, sobald ein fremdes Feld gefunden oder ein Sieg festgestellt wird. So kann in diesem Beispiel die Horizontale mit maximal vier Zugriffen auf das Spielfeld analysiert werden. Dasselbe Prinzip funktioniert auch in der vertikalen und in beiden diagonalen Richtungen. Somit ist es möglich, einen Sieg innerhalb von maximal 16 Zugriffen zu erkennen. Diese Methode ist sehr performant, verglichen mit einer Suche über das komplette Feld und spart so sehr viel Rechenzeit. Ich kann es mir also leisten, diese Methode beim erstellen jeder einzelnen Node aufzurufen, um zu prüfen, ob die Node Kinder haben sollte. Wenn die Node einen Sieg beschreibt, macht es keinen Sinn, den Baum unter ihr weiter zu berechnen. So wird im Endeffekt auch der Baum kleiner.

Performance Messungen

Die Performance ist im optimierten Zustand (Multithreading + Pruning) akzeptabel. Man kann gegen den Algorithmus spielen, ohne besonders lange warten zu müssen. Trotzdem ist es sehr schwer, gegen ihn zu gewinnen. Ich habe zwei verschiedene Applikationen erstellt und abgegeben, eine davon arbeitet mit einer Suchtiefe von 6, die andere mit einer Tiefe von 7. Die Version mit der niedrigeren Suchtiefe ist natürlich sehr viel schneller, und der Algorithmus antwortet beinahe sofort. Bei der Version mit der höheren Tiefe muss man besonders am Anfang des Spiels kurz warten.

Ich habe beim Testen der Anwendung folgende Daten gemessen:

Dauer der Berechnung des ersten Zuges:

Suchtiefe	**Keine Optimierung**	**Multithreading**	**Pruning**	**Multithreading + Pruning**
6	2585 ms	856 ms	578 ms	475 ms
7	17911 ms	6639 ms	3356 ms	2743 ms
8	160190 ms	39860 ms	21142 ms	18973 ms

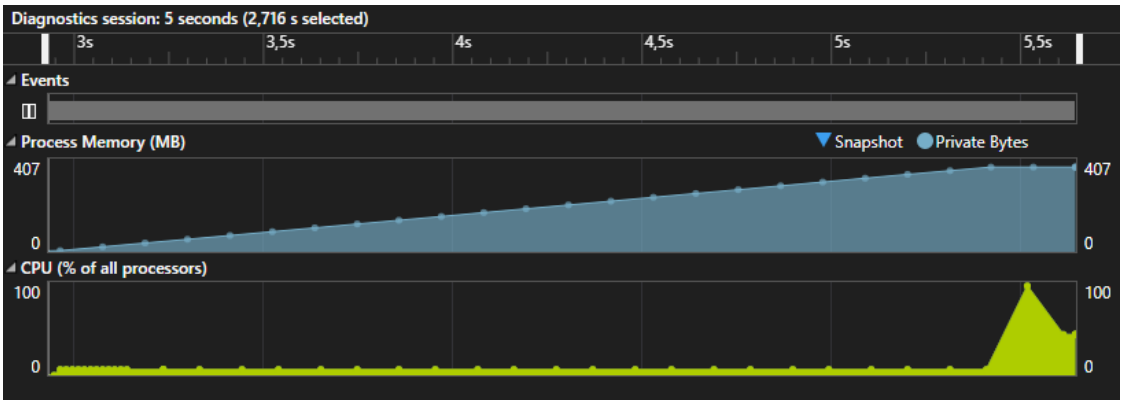
Maximaler RAM Speicherverbrauch der Anwendung:

Suchtiefe	**Keine Optimierung**	**Multithreading**	**Pruning**	**Multithreading + Pruning**
6	55 MB	55 MB	55 MB	55 MB
7	369 MB	370 MB	369 MB	370 MB
8	~ 2,5 GB	~ 2,5 GB	~ 2,5 GB	~ 2,5 GB

An diesen Messungen kann man erkennen, dass sowohl das Pruning als auch das Multithreading die Rechenzeit extrem verbessert. Im finalen Projekt werde ich also beides aktivieren.

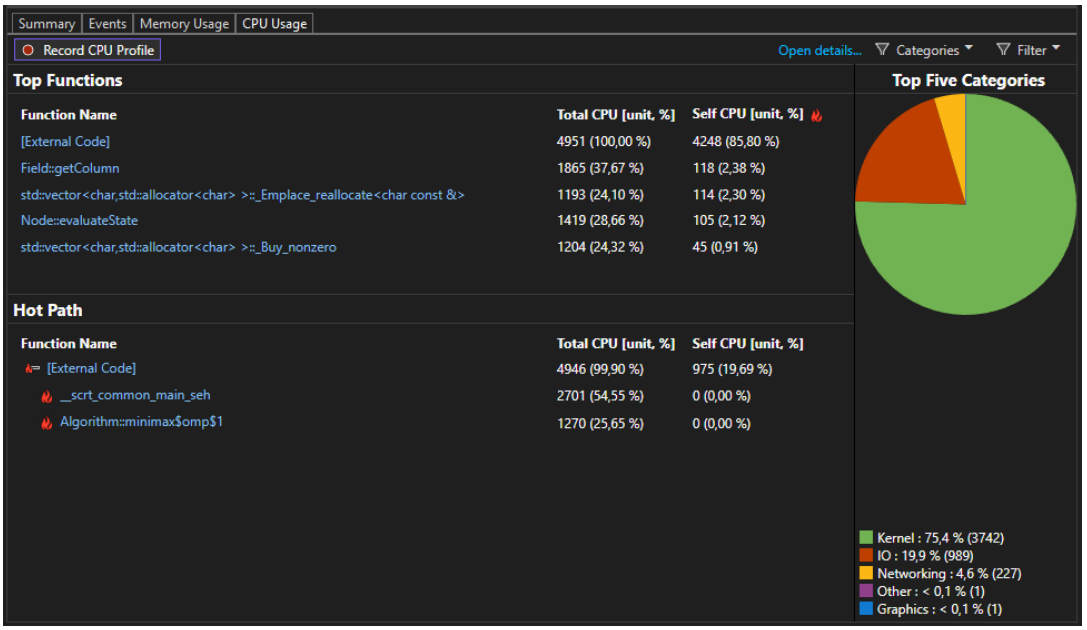
Die Optimierungen haben jedoch nicht wirklich eine Auswirkung auf den Speicherverbrauch. Das liegt daran, dass alle Optimierungen beim Bewerten des Suchbaums stattfinden und nicht beim Erstellen. Ich habe versucht das Erstellen des Suchbaumes ebenfalls zu multithreaden, es gab jedoch Probleme mit den Zugriffen aus den verschiedenen Threads, da alle Nodes in einer Liste sind, die ein Member einer anderen Node sind. Diese Node könnte jedoch in einem anderen Thread sein.

Das folgende Bild zeigt eine Messung mit einer Suchtiefe von 7. In diesem Test waren sowohl das Multithreading als auch das Pruning aktiviert.



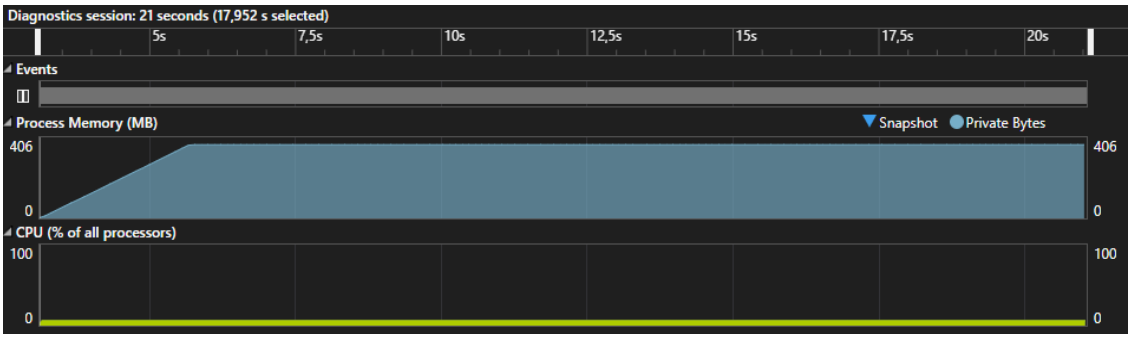
Der Graph zeigt die Speicher- und CPU-Auslastung des Prozesses. Der sichtbare Bereich wurde aufgenommen, während die Funktion Algorithm::getNextMove() läuft. Es zeigt also den Ressourcenverbrauch des Suchbaumes und der Minimax-Funktion. Da diese Daten mitten im laufenden Prozess aufgenommen wurden, beginnt die Zeitleiste oben nicht bei 0, sondern erstreckt sich über die 2716 ms, die die Funktion gebraucht hat, um zu einem Ergebnis zu kommen.

Man kann ablesen, dass der längste Prozess das Erstellen des Suchbaumes ist. Das liegt daran, dass nur sehr wenige Optimierungen zum Erstellen implementiert sind. Das Auswerten des Baumes (Der Teil, in dem der Speicherverbrauch konstant bleibt) geschieht verglichen dazu sehr schnell.

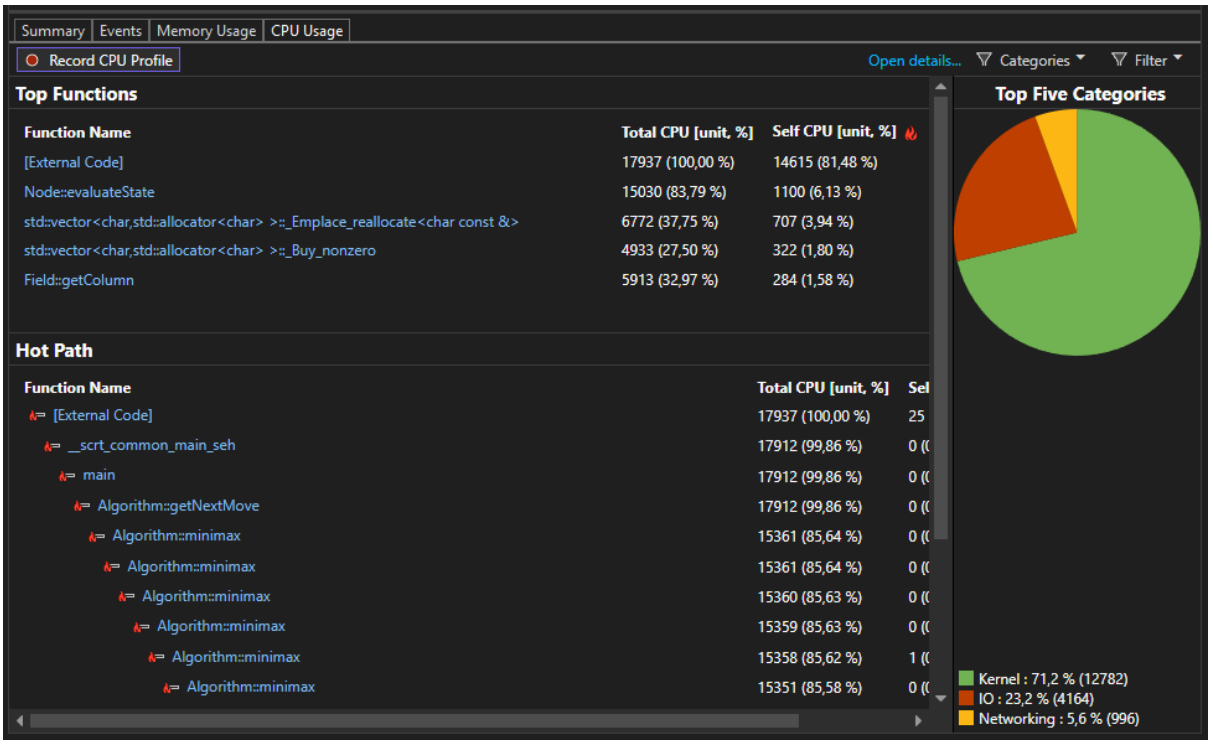


Ein genauerer Blick auf die Analyse zeigt, dass Funktionen wie Field::getColumn oder das neu allokieren von Speicher für die Spielfeldmatrizen zusammen über 50% der CPU Operationen stellen. Diese Funktionen müssten also optimiert oder vermieden werden, um den Prozess weiter zu beschleunigen.

Zum Vergleich hier eine Messung mit einer Tiefe von 7, jedoch ohne Multithreading und ohne Pruning. Achtung, die Skalierung der Zeitachse ist nicht dieselbe.
Die Zeit die für das Aufstellen des Baumes benötigt wird (Rampe im Speicherverlauf), ist vergleichbar mit der vorherigen Messung, das Auswerten des Baumes danach dauert jedoch wesentlich länger.



Das CPU-Profil zeigt auch, dass nun knapp 84% der Zeit innerhalb der Bewertungsfunktion Node::evaluateState() verbracht wird. Dies liegt daran, dass nun wirklich alle Endpunkte bewertet werden und dass die Bewertungsfunktion nicht sonderlich performant ist.



Fazit

Der Algorithmus schaut 6 bzw. 7 Schritte weit in die Zukunft.
Die Dauer der Berechnung ist für mich akzeptabel und ich habe es nicht einmal geschafft, gegen ihn zu gewinnen (vielleicht bin ich auch einfach schlecht in diesem Spiel).

Trotzdem gibt es noch Verbesserungsmöglichkeiten, vor allem bei der Größe des Baumes und dessen Speicherverbrauch. Ein paar Ideen, die Ich noch habe sind:

- Momentan werden z.B. keine Symmetrien innerhalb des Baumes berücksichtigt, da ich das nicht fehlerfrei hinbekommen habe. Das würde die Größe des Suchbaumes noch einmal stark einschränken.
- Eine weitere Idee betrifft das Speichern der Spielfelder. Eine Vektormatrix ist leider sehr verschwenderisch, da 8 Bit pro Zelle gespeichert werden, obwohl wir eigentlich nur zwischen drei Werten unterscheiden (“X”, “O”, “ “). Eine Vereinfachung des Spielfeldes würde auch dazu führen, dass der Baum schneller aufgebaut werden kann.
- Die momentane Bewertungsfunktion ist nicht besonders performant, da mir keine besonders schnelle Methode eingefallen ist, auf einem Feld nach Paaren gleicher Spielsteine zu suchen, ohne relativ unperformant über das ganze Feld zu iterieren.
- Wenn das Erstellen des Baumes auf mehrere Threads verteilt werden könnte, würde das sicherlich auch noch einmal Zeit sparen.

Trotzdem bin ich mit dem Ergebnis zufrieden, da es seinen Zweck erfüllt und ein schwieriger Spielgegner ist.

Quellen und Referenzen

OpenMp:

- Ausführliche Beschreibung: https://de.wikipedia.org/wiki/OpenMP
- Website: https://www.openmp.org
- Doku für die Version, die ich genutzt habe: https://learn.microsoft.com/de-de/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170

Quellcode:

- https://github.com/LouisWatermeyer/KI-Project

Fertige Anwendung:

- https://github.com/LouisWatermeyer/KI-Project/releases/tag/V1.0